Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 156 (2006) 135–150

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Prototyping SOS Meta-theory in Maude

MohammadReza Mousavi[1] Michel A. Reniers[2]

Department of Computer Science, Eindhoven University of Technology (TU/e), P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Abstract

We present a prototype implementation of SOS meta-theory in the Maude term rewriting language. The prototype defines the basic concepts of SOS meta-theory (e.g., transition formulae, deduction rules and transition system specifications) in Maude. Besides the basic definitions, we implement methods for checking the premises of some SOS meta-theorems (e.g., GSOS congruence meta-theorem) in this framework. Furthermore, we define a generic strategy for animating programs and models for semantic specifications in our meta-language. The general goal of this line of research is to develop a general-purpose tool that assists language designers by checking useful properties about the language under definition and by providing a rapid prototyping environment for scrutinizing the actual behavior of programs according to the defined semantics.

Keywords: Formal Semantics, Structural Operational Semantics, Term Rewriting, Language Prototyping, Maude.

1 Introduction

Defining a formal semantics for a language is usually among the very first steps of bringing it into the formal world. The process of defining the semantics involves many choices some of which are very implicit and hidden from the designer's naked eyes. Furthermore, there is usually no reference point to check whether the end result is "correct" and the right choices have been made during the process of defining the semantics. For a complicated language, it soon goes beyond human capabilities to keep track of the consequences of design-decisions in the semantics and one can often overlook possible counter-intuitive phenomena introduced there. Proving theorems about

[1] Email: M.R.Mousavi@tue.nl

[2] Email: M.A.Reniers@tue.nl

doi:10.1016/j.entcs.2005.09.030

intuitive properties and checking several instances of system runs (according to the given semantics) against the intuition are good ways to build insight and confidence about the semantics.

In this paper, we report an initial attempt to implement a general-purpose tool that provides a language designer with the above possibilities. Our prototype is geared to Structural Operational Semantics (SOS) [22] which is by now a de facto standard in defining semantics for specification and programming languages.

There has been a reasonable body of knowledge developed around the concept of SOS which aims at proving useful properties about SOS specifications [3]. Congruence results [13,5], deriving equational theories [2] and conservative extensions [11] are among the most notable meta-theories in this direction. We aim at defining a framework which allows us to check the premises of simple instances of the above meta-theorems for SOS specifications and further allows for animating specifications and programs according to the given semantics. The Maude term rewriting language [1] comes in very handy as the base language for our implementation.

The rest of this paper is structured as follows. In Section 2, we review the related work in prototyping SOS languages and proving meta-theorems about them. Afterwards, in Section 3, the general definition of Transition System Specification as a formalization of SOS is presented and its implementation in Maude is described. An instance of a congruence meta-theorem is then defined in Section 4 and implemented. Section 5 defines a simple operational conservativity theorem and illustrates its implementation. Section 6 is devoted to animating SOS specifications. Finally, Section 7 concludes the paper and proposes several possible extensions of our prototype.

2 Related Work

Despite the bulk of knowledge in the area of SOS meta-theory, little has been done in implementing them. In [21], we report our initial experiment with implementing an instance of SOS specification in the Maude rewriting logic [1] which was used as a prototype simulation and model checking environment for the particular target language. This initial prototype helped us check and remove a few "bugs" in our initial semantics. Apart from our previous implementation, other authors have studied the, rather clear, link between the rewriting logic [15] and SOS [22] both from a theoretical [10,15,16,17] as well as from a practical point of view [8,9,24,26,27].

In [8], the outline of a translation from Modular SOS (MSOS) [18,19] to the Maude rewriting logic is given and proven correct. The translation is quite

straightforward and the main technical twist is in the decomposition of labels into the configurations in the source and the targets of the transitions which is due to the structure of labels in MSOS. The translation is fully implemented and details of this implementation can be consulted in [7]. The main difference between this research and ours stems from the fact that we take SOS as our point of departure and this may help us benefit from its theoretical history (e.g., the meta-theorems implemented in this paper) and practical popularity.

Verdejo in [24] and Verdejo and Marti-Oliet in [26,27] report the implementation of a number of instances of SOS semantics in Maude. Our approach is very close in essence to their work in that SOS deduction rules are interpreted as Maude conditional rewrite rules. This approach is referred to as *transitions as rewrites*. We contribute to their work by first, raising the level of abstraction a bit so that one can talk about SOS rules in general, specify and execute them and reason about them and second, we implement a bit more involved case of SOS with negative premises in our framework.

Earlier versions (of Maude) did not support conditional rewriting with rewrites as conditions. Thus, a different approach has been proposed in [15] to implement SOS, called *transitions as judgements*. In this approach each transition is implemented as a term and SOS deduction rules are implemented as rewrite rules that rewrite the transition in the conclusion to the transitions in the premises or vice versa (i.e., constructing a proof structure using a bottom-up or a top-down approach). Both of these approaches have been suggested by [15] and the former has been implemented in [25]. Both transitions as rewrites and as judgements can be useful. In [26], it is reported that the transitions as rewrites approach is easier to implement and causes less complications. Furthermore, modeling transitions as rewrites allows for exploiting available search and model checking libraries implemented in Maude to investigate the behavior of a model.

LETOS [14] is a tool that generates LaTex documents as well as executable animation code in Miranda [23] from a wide range of semantics, including some forms of SOS. A first attempt to implement an SOS meta-theorem, concerning operational conservativity of [13] is also reported in [14]. However, the implementation does not fully check this theorem and only checks the *source-dependency* requirement which is one of the hypotheses of the conservativity theorem of [13].

3 Transition System Specifications

Motivation

Transition System Specification (TSS) is a formalization of SOS proposed in [13]. In this section, we define the concept of TSS with constant labels and negative premises [12,5] and formalize it in Maude. A natural extension of our framework would be to support terms as labels.

Basic Definitions

We assume a countably infinite set of *variables* $V = \{X_0, Y_0, \ldots\}$. A *signature* Σ contains a number of function symbols (composition operators: $f, g, \ldots$) with fixed arities $(ar(f), ar(g), \ldots)$. Function symbols with arity 0 are called constants. The set of process terms, denoted by $\mathcal{T}(\Sigma, V)$ with typical members s, t, s', t', is inductively defined on a signature Σ and a set of variables V. A substitution σ replaces variables in a term with other terms. A term s *can match* (is unifiable with) term t if there exists a substitution σ such that $\sigma(t) = s$. The set of variables appearing in term t is denoted by $vars(t)$. Term t is closed if $vars(t) = \emptyset$.

A *Transition System Specification (TSS)* is a tuple (Σ, V, L, D) where Σ is a signature, V is a set of variables, L is a set of labels and D is a set of deduction rules. For all $l \in L$, and $s, s' \in \mathcal{T}(\Sigma, V)$ we define that $s \xrightarrow{l} s'$ and $s \stackrel{l}{\nrightarrow}$ are positive and negative formulae, respectively. We refer to s as the *source* of both formulae and s' as the *target* of the positive one. A deduction rule $dr \in D$, is defined as a tuple (H, c) where H is a set of formulae and c is a positive formula. The formula c is called the *conclusion* and the formulae from H are called *premises*. A rule with an empty set of premises is called an axiom. A deduction rule (H, c) is denoted by $\frac{H}{c}$. The notion of substitution and matching are lifted to formulae as expected.

Formalization in Maude

Labels and variables are defined as sorts `Labels` and `Vars`, respectively. Elements of sort `Labels` are left to be defined by the user, but we treat the labels as constants (possibly with some algebraic structure). Basic constructors `X- n` and `Y- n` are defined for variables X_n and Y_n indexed by natural numbers. A signature is to be defined per specification. Function symbols in the signature are to be defined using the Maude syntax. For example, a binary operator $\_ + \_$ can be defined as `op _ + _ : T T -> T [ctor]`, where `T` is the given name for the sort of terms and `ctor` stands for constructor. Substitutions and matching are already defined for variables and have to be lifted by the user to the term level. We foresee the possibility of generating

substitution and matching axioms automatically by examining the signature at meta-level.

Formulae $s \xrightarrow{l} s'$ and $s \xrightarrow{l} \not\rightarrow$ are denoted by expressions `s == l ==> sp` and `s == l =/=>`, respectively. A TSS is a functional theory parameterized by the signature, variables and labels. However, due to a technical issue (lack of support for parameterized modules in `upModule` method) in the current implementation of Maude, we implement them as plain functional modules. Transforming our implementation to the parameterized setting is a matter of renaming interfaces and sort names. A deduction rule $\frac{H}{c}$ is denoted by `H === c` and deduction rules in a set are separated by commas.

Using the general implementation of TSS's and related concepts, we can specify instances of SOS specification as shown in the examples given below. Note that the examples are there for explanation purposes and do not necessarily stand for practical and meaningful instances of SOS.

Examples

Table 1 shows the SOS specification of a Minimal Process Algebra (MPA) in our framework. The Maude code is self-explanatory and is almost the same as the text appearing in any SOS specification. The signature consists of a constant `delta` for the deadlocking process, a class of unary operators `a ; _` for action prefixing with `a` being a member of the sort `BAct` (for Basic Actions) and a binary operator `_ + _` for the non-deterministic choice. The concepts of substitution, matching and variables of a term are defined by a simple structural induction on terms (the base cases for these definitions are defined generically in the module `Term-Match`). Deduction rules define the operational semantics of action prefixing and non-deterministic choice.

Our next example is a simple extension of MPA with the aspect of timing presented in Table 2. In this extension, we have a new label `tick` for the time transition and a new unary operator `delay ; _` which causes a time transition to happen. Apart from the deduction rules specified before, we have to add deduction rules defining the behavior of the delay operator and also the time-deterministic nature of choice (cf. [4]), i.e., time will only decide about non-deterministic choice if one of the two components blocks the time transition.

To simplify matters in the remainder, we assume that TSS's extending other specifications import (`include`) the theory to be extended but have all the newly introduced function symbols, labels and deduction rules in a single module.

```
fmod MPA-TSS is
 inc Term-Match .
 inc TSS-Definition .
 sort BAct .
 subsort BAct < Labels .
*** MPA Signature
 op delta   : -> T [ctor] .
 op _ ; _ : BAct T -> T [ctor] .
 op _ + _ : T T -> T [ctor] .
*** Substitutions and Matching
 op a         : -> BAct [ctor] .
 var act      : BAct .
 var sigma    : Sbst .
 vars s t sp tp  : T .
 eq sigma ( delta ) = delta .
 eq sigma (act ; s) =
    act ; sigma (s) .
 eq sigma (s + t) =
    sigma (s) + sigma (t) .
 eq vars (delta) = emptyVars .
 eq vars (act ; s) = vars (s) .
 eq vars (s + t) =
    vars (s) cup vars (t) .
 eq match (delta, delta) = emptySbst .
 eq match ((s + t), (sp + tp) ) =
    (match (s, sp), match (t, tp)) .
eq match ((act ; s), (act ; t)) =
   match (s, t) .
*** Operational Semantics of MPA
 op MPA  : -> TSS .
 eq MPA =
 ( ===
   a ; X-  0 == a ==> X-  0 ) ,
 ( X- 0 == a ==> Y-  0
   ===
   X- 0 + X- 1 == a ==> Y- 0 ) ,
 ( X- 1 == a ==> Y-  1
   ===
   X- 0 + X- 1 == a ==> Y-  1 ) .
endfm
```

Table 1
Structural Operational Semantics of MPA in Maude

```
fmod MPAT-TSS is
 inc MPA-TSS .
 op tick       : -> Labels [ctor] .
 op delay ; _ : T -> T [ctor] .
 eq sigma (delay ; s)  =
    delay ; sigma (s) .
 eq match ((delay ; s),
    (delay ; t)) = match (s , t) .
 eq vars (delay ; s) = vars (s) .
*** Operational Semantics of MPAT
 op MPAT  : -> TSS .
 eq MPAT =  ( MPA,
 ( ===
   delay ; X- 0 == tick ==> X- 0 ),
 (( X- 0 == tick =/=> ,
    X- 1 == tick ==> Y-  1 )
    ===
   X- 0 + X- 1 == tick ==> Y-  1 ) ,
 (( X- 0 == tick ==> Y- 0 ,
    X- 1 == tick =/=> )
    ===
    X- 0 + X- 1 == tick ==> Y-  0 ) ,
 (( X- 0 == tick ==> Y- 0 ,
   X- 1 == tick ==> Y- 1 )
    ===
    X- 0 + X- 1 == tick ==>
                  Y-  0 + Y-  1 )) .
 endfm
```

Table 2
A Simple Extension of MPA with Time in Maude

4 A Congruence Meta-Theorem

Motivation

Operational semantics usually induces a labelled transition system for closed terms and it is interesting to observe when two terms show the same behavior. This notion of *behavioral equivalence* can be used to establish that an implementation is correct with respect to its specification. It is very much desired for a notion of behavioral equivalence to be compositional or in tech-

nical terms to be a *congruence*. Several congruence meta-theorems have been proposed in the literature [3]. These meta-theorems guarantee that a notion of behavioral equivalence on a particular semantics is a congruence if the deduction rules of the semantics conform to some syntactic criteria. Here, as an example, we choose a practically useful instance of such meta-theorems from [5] and implement it in our framework. The meta-theorem that we implement guarantees that the notion of (strong) *bisimilarity* for a particular TSS is a congruence.

Basic Definitions

If TSS *tss proves* a transition formula $p \xrightarrow{l} p'$, we denote it by $tss \vDash p \xrightarrow{l} p'$ (see Section 6 for the precise meaning of "proving a transition"). Given a TSS *tss*, a symmetric relation R on closed terms is a *bisimulation* relation when $\forall_{p,q}\ (p,q) \in R \Rightarrow \forall_{l,p'}\ tss \vDash p \xrightarrow{l} p' \Rightarrow \exists_{q'}\ tss \vDash q \xrightarrow{l} q' \wedge (p',q') \in R$. Two closed terms p and q are *bisimilar*, denoted by $p \leftrightarrows q$, when there exists a bisimulation relation R such that $(p,q) \in R$.

An equivalence relation R on terms is a congruence when for all function symbols $f \in \Sigma$ and for all terms $p_i, q_i \in \mathcal{T}(\Sigma, V)$ $(0 \leq i < ar(f))$, if $(p_i, q_i) \in R$ for all $0 \leq i < ar(f)$, then $(f(p_0, \ldots, p_{ar(f)-1}), f(q_0, \ldots, q_{ar(f)-1})) \in R$.

A deduction rule is in *GSOS format* [5] when it has the following form:

$$\frac{\{x_i \xrightarrow{l_{ij}} y_{ij} \mid i \in I, j \leq m_i\} \cup \{x_j \overset{l_{jk}}{\nrightarrow} \mid j \in J, k \leq n_j\}}{f(x_0, \ldots, x_{ar(f)-1}) \xrightarrow{l} t},$$

where f is a function symbol in the signature, x_i and y_{ij}'s are all distinct variables, I and J are subsets of $\{0, \ldots, ar(f) - 1\}$, m_i and n_j are natural numbers and $vars(t) \subseteq \{x_i, y_{jk} | i \in I \cup J, j \in I, k \leq m_i\}$. A TSS is in GSOS format when all its deduction rules are.

Theorem 4.1 (Congruence for GSOS [5]) *If a TSS is in GSOS format then bisimilarity (w.r.t. its induced transition relation) is a congruence.*

Formalization in Maude

Our formalization of the GSOS format makes use of the reflective semantics of Maude. Reflection in this context means that any rewrite theory can be interpreted as an object inside a "universal" rewrite theory. This way one can look at theories from a meta-level viewpoint and reason about them. Using this capability we examine the structure of deduction rules by first, automatically compiling a list of function symbols in the signature (with a target source `T`) using the meta-level operation `getOps` and then, checking whether

```
fmod Test-TSS is
 inc Term-Match .
 inc TSS-Definition .
*** Signature
 ops a b  : -> T [ctor] .
 op ft _ : T -> T [ctor] .
 op l : -> Labels [ctor] .

*** Operational Semantics
 op Test  : -> TSS .
 eq Test =
  ( ===
    ft (a) == l ==> ft(a)) .
endfm
```

Table 3
A Simple TSS Violating GSOS Format

the premises contain only the right kind of variables in their source and target. Using meta-level functions, our implementation becomes independent from the choice of signature and the set of defined and used variables.

The implemented `GSOS-Check` method takes the name of the TSS (of type `Qid`) as a parameter, reads the signature of the TSS from the corresponding functional module, checks the conformance of rules and outputs a string which states the positive result, or alternatively, outputs one deduction rule which does not conform to the GSOS format.

Examples

Consider the TSS's of MPAT given in Table 2. The following statements show how to check conformance of MPAT to GSOS format and the outcome of this check (applying a similar command on MPA results in a similar result).

```
Maude> red in GSOS-Check : GSOS-Chk ( 'MPAT-TSS , MPAT ) .
reduce in GSOS-Check : GSOS-Chk('MPAT-TSS,MPAT) .
rewrites: 211 in 30ms cpu (80ms real) (7033 rewrites/second)
result Message: successmsg
("GSOS-Check: TSS conforms to GSOS.")
```

Now, consider the TSS shown in Table 3. Applying GSOS-Check on this TSS results in the following error messages.

```
Maude> red in GSOS-Check : GSOS-Chk ( 'Test-TSS , Test ) .
reduce in GSOS-Check : GSOS-Chk('Test-TSS,Test) .
rewrites: 49 in 0ms cpu (0ms real) (~ rewrites/second)
result Message: errormsg(
"GSOS-Check: Error, the following rule:",
emFr === ft(a)  == l ==> ft(a),
"has more than one operator in its source of conclusion.")
```

The GSOS meta-theorem only provides sufficient (and not necessary) conditions for the congruence of bisimilarity but in the above case, bisimilarity is indeed not a congruence: $a \underline{\leftrightarrow} b$ since both of them have no operational behavior but it does not hold that $f(a) \underline{\leftrightarrow} f(b)$ since the former can make a transition using the rule mentioned above while the later cannot make any transition.

5 Operational Conservativity

Motivation

Using the concept of functional modules, we can include SOS specifications in new modules and extend them with new function symbols, new labels and new deduction rules. It is often crucial to make sure that such extensions are *conservative*, i.e., they do not change the operational behavior of old terms. In this section, we formulate a simple instance of operational conservativity meta-theorems (by restricting it to the GSOS framework) and explain its implementation in Maude.

Basic Definitions

To extend a language defined by a TSS, one may have to combine an existing signature with a new one. However, not all signatures can be combined into one as the arities of the function symbols may clash. To prevent this, we define two signatures to be *consistent* when they agree on the arity of the shared function symbols.

Consider consistent TSS's $tss_0 = (\Sigma_0, V_0, L_0, D_0)$ and $tss_1 = (\Sigma_1, V_1, L_1, D_1)$. The extension of tss_0 with tss_1, denoted by $tss_0 \cup tss_1$, is defined as $(\Sigma_0 \cup \Sigma_1, V_0 \cup V_1, L_0 \cup L_1, D_0 \cup D_1)$. If $\forall_{p \in C(\Sigma_0)}\ \forall_{p' \in C(\Sigma_0 \cup \Sigma_1)}\ \forall_{l \in L_0 \cup L_1}\ tss_0 \cup tss_1 \vDash p \xrightarrow{l} p' \Leftrightarrow tss_0 \vDash p \xrightarrow{l} p'$, then $tss_0 \cup tss_1$ is an *operationally conservative extension* of tss_0.

The following theorem is a simplification of the general theorem presented in [11].

Theorem 5.1 (Operational Conservativity for GSOS) *Consider consistent TSS's $tss_0 = (\Sigma_0, V_0, L_0, D_0)$ and $tss_1 = (\Sigma_1, V_1, L_1, D_1)$ both in the GSOS format. $tss_0 \cup tss_1$ is an operationally conservative extension of tss_0 if for all deduction rules $d \in D_1$, one of the following conditions holds:*

(i) *d has a function symbol from $\Sigma_1 \setminus \Sigma_0$ in the source of conclusion, or*

(ii) *d has a positive premise $x_i \xrightarrow{l_{ij}} y_i$ with $l_{ij} \in L_1 \setminus L_0$.*

Formalization in Maude

Formalization of this meta-theorem goes along the same lines as that of the congruence meta-theorem. First, we compile a list of function symbols and labels in the extended and extending TSS's and then check the deduction rules of the extending TSS to either include a fresh function symbol in the source of conclusion or a fresh label in at least one of the positive premises.

Example

Checking the conservativity of the extension of MPA (Table 1) with time (Table 2) goes as follows.

```
Maude> red in CONSV-Check : Cons-Chk ( 'MPA-TSS , MPA, 'MPAT-TSS, MPAT ) .
reduce in CONSV-Check : Cons-Chk('MPA-TSS,MPA,'MPAT-TSS,MPAT) .
rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
result Message: successmsg
("CONS-Check: Operational conservativity theorem checked successfully.")
```

Trying the same routine on a non-conservative extension results in an error message which points out a deduction rule and a hypothesis of the conservativity theorem that has been violated.

6 Animating SOS

Motivation

Despite their operational nature, SOS specifications are not in general executable. As shown in [5], slight extensions to GSOS easily ruin the decidability of proving a transition. To add to the complications, it was shown in [12] that not all SOS specifications are meaningful, in that they may not define a transition relation at all or they may ambiguously allow for more than one transition relation. By taking GSOS as a framework, one may be relieved of these hassles. Our animation method does not require the TSS to be in GSOS. However, it guarantees to terminate and produce a sound result if the TSS is *strictly and finitely stratified* (GSOS specifications are among these). Next, we precisely define what it means for a transition to be provable from a TSS and how this concept is formalized in Maude.

Basic Definitions

A positive closed formula ϕ is *provable* from a set of positive formula T and a transition system specification tss, denoted by $(T, tss) \vdash \phi$, if and only if there is a well-founded upwardly branching tree of which the nodes are labelled by closed formulae such that

- the root node is labelled by ϕ, and
- if the label of a node q, denoted by ψ, is a positive formula and $\{\psi_i \mid i \in I\}$ is the set of labels of the nodes directly above q, then there is a deduction rule $\dfrac{\{\chi_i \mid i \in I\}}{\chi}$ in tss (N.B. χ_i can be a positive or a negative formula) and a substitution σ such that $\sigma(\chi) = \psi$, and $\sigma(\chi_i) = \psi_i$ for all $i \in I$;
- if the label of a node q, denoted by $s \overset{l}{\nrightarrow}$, is a negative formula then there

exists no s' such that $s \xrightarrow{l} s' \in T$.

A *stable model*, also called a transition relation, defined by a transition system specification tss is a set of formulae T such that for all closed positive formulae ϕ, $\phi \in T$ if and only if $(T, tss) \vdash \phi$. A transition $s \xrightarrow{l} s'$ is provable from tss, denoted by $tss \vDash s \xrightarrow{l} s'$ when tss induces a unique stable model T and $s \xrightarrow{l} s' \in T$.

One way to make sure that a TSS defines a unique transition relation is through the following notion of stratification.

A *stratification* of a transition system specification tss is a function $\mathcal{S}$ from closed positive formulae to an ordinal such that for all deduction rules of tss of the following form:

$$\frac{\{t_i \xrightarrow{l_i} t'_i \mid i \in I\} \quad \{t_j \xrightarrow{l_j} \mid j \in J\}}{t \xrightarrow{l} t'}$$

such that for all closed substitutions σ we have (1) for all $i \in I$, $\mathcal{S}(\sigma(t_i \xrightarrow{l_i} t'_i)) \leq \mathcal{S}(\sigma(t \xrightarrow{l} t'))$ and (2) for all $j \in J$ and t'', $\mathcal{S}(\sigma(t_j \xrightarrow{l_j} t'')) < \mathcal{S}(\sigma(t \xrightarrow{l} t'))$. A stratification measure is *strict* if in the previous definition $\leq$ is replaced by $<$. A stratification measure is *finite* if the range of it is the natural numbers. A transition system specification is called *(strictly and/or finitely) stratified* if and only if there exists a (strict and/or finite) stratification function for it.

Corollary 6.1 *A TSS in GSOS is strictly and finitely stratified. Thus, all TSS's conforming to the GSOS format have a unique stable model [6].*

Formalization in Maude

We interpret deduction rules as conditional rewrite rules. In order to check for possible transitions for a closed term s, we first look for a deduction rule $d \in tss$ of the form $\frac{H}{s' \xrightarrow{l} t}$ such that s' can match (i.e., is unifiable with) s. The unification of s' with s results in a substitution σ_0 evaluating the variables of s. We aim at completing σ_0 into σ such that first, σ evaluates all the variables in d (thus, the variables in t), second, all positive premises evaluated by σ are provable from tss and finally, negative premises evaluated by σ cannot be contradicted by a proof from tss. To this end, we examine the premises in the following order.

We search for premises of d of which its source is evaluated by the substitution σ_j constructed so far.[3] If the premise is a negative one, we make sure

[3] In a large class of TSS's such a premise can be found. Such TSS's are theoretically

that this fully evaluated premise cannot be contradicted by a proof from tss. If it is a positive premise of the form $t_i \xrightarrow{l_i} t'_i$, we try to construct a proof for a transition of $\sigma_j(t_i)$ to evaluate the variable in t'_i. If we succeed in constructing such a proof, we add the valuation of the variable in t'_i to σ_j resulting in σ_{j+1}. This process continues until no premise remains to be examined.

Each of the above mentioned steps is implemented as a conditional rewrite rule, rewriting a set of premises and a partial substitution to a (possibly more complete) substitution. The transition of term s is then modeled as a conditional rewrite rule from $\sigma_0(s)$ to $\sigma_n(t)$ where σ_n results from the rewrite rules of the procedure described above. For pure TSS's [13] such a substitution evaluates all variables in t (the target of the transition). For non-pure TSS's, variables in t that are not evaluated by the above procedure are mapped in σ_n to an arbitrary closed term (again using a rewrite theory).

Next, we quote an excerpt of the Maude code implementing this procedure.

```
crl ( tss  |- ( s == l ==> ) ) =>
    ( ( sigma, rho ) ( target ( conc ( rule ))))
if
  ( rule , tssp )  := tss                          /\
  sigma := match (  conc(rule), ( s == l ==> ))    /\
  ( tss ||- ( sigma (prem(rule)))) =>  rho         /\
  ( ( sigma , rho ) :: Sbst )                      /\
  ( closed ( (sigma, rho) (target (conc(rule) ))))  .
```

In the above code, `crl` stands for conditional rewriting which rewrites the term before the arrow `=>` to the term after provided the condition specified by the `if` clause holds. In this case, the term before the rewrite arrow consists of the TSS under consideration (`tss`) the source (`s`) and the label (`l`) of the transition. The term after the rewriting arrow is the target of the conclusion of the matching deduction rule (`rule`) with the substitution (`sigma, rho`) to be constructed by the above mentioned procedure applied to it. In the condition part of the rewrite rule, first, pattern matching is used to pick an arbitrary rule from the TSS. Then, it is checked whether there is a substitution `sigma` matching the source of the rule and `s`. Next, it is checked whether a substitution `rho` can be constructed so that the premises of the rule can be satisfied (to be explained further in the remainder). If such a substitution can be found and it evaluates all variables in the target of the conclusion of `rule`. Here, we omit the case (of non-pure thus, non-GSOS rules) where the resulting substitution does not evaluate all variables in the target of the conclusion of `rule`.

We distinguish the following two cases for checking the premises of the

characterized as *pure and well-founded* TSS's [13]. For TSS's that do not have such property, a premise is chosen arbitrarily and different closed substitutions for its source are examined.

deduction rule. If the premise is a positive one, the the check is nothing but looking for a transition from the source which matches the target of the premise. The matching substitution `sigma` is then used to evaluate the rest of the premises.

```
crl ( tss ||- ( s == l ==> t ) )  => sigma
if
  ( tss |- ( s == l ==> ) )  => sp /\
  sigma := match ( t, sp )           .
```

If the premise is a negative one, we use the meta-level method `metaRewrite` to check whether a contradicting rewrite (transition) can be found using the same rewriting theory. Note that the check for negative premises does not add any information with respect to the substitution under constructed. Thus, the result of the rewrite is the empty substitution (`emSbst`). Again in both of these cases, we omit the rewrite rules dedicated to the cases where the chain of premises is broken (i.e., the rule is not pure) and no transition with a closed source can be found among the evaluated premises.

```
crl ( tss ||- (s == l =/=> ) )  => ( emSbst  )
if
  possible? := metaRewrite( ['TSS-Animation],
                            upTerm( ( tss |- ( s == l ==> ) ) ), 1 )  /\
  (possible? :: ResultPair )                          .
```

The above procedure, upon termination, gives us a complete proof for the transition with a guarantee that negative premises cannot be refuted using our rewrite theory, thus, using the SOS semantics. However, in general this procedure need not terminate. Consider the following two SOS specifications.

```
a == l ==> a              a == l =/=>
===                       ===
a == l ==> a              a == l ==> a
```

The Maude tool crashes when trying to animate any of the above two TSS's since the procedure results in an infinite chain of rewrites each being a condition for the next. However, this problem does not occur in GSOS specifications and in general, strictly and finitely stratified TSS's because for such TSS's checking conditions of each rewrite results in a condition with a lower stratification measure. Hence, the depth of conditional rewrite checks for a transition is always finite. Also, breadth of this search is always finite, since we can only specify a finite number of rules each having a finite number of premises. If the proof search is successful on all premises, it provides us with a substitution that valuates the variables in the target of the deduction rule and hence, we are able to find a possible transition for term s using the label and evaluated target of the conclusion of the deduction rule.

It is worth mentioning that this procedure is non-deterministic in that there

may be several provable transitions for a closed term s. The Maude semantics has an inherent support for non-deterministic rewrite theories and hence the choice among such transitions remains non-deterministic and is eventually made by the Maude rewriting engine. Using the Maude tool one can browse through provable transitions, check for provability of a particular transition and even use logical formulae (Linear Temporal Logic (LTL) formulae) to model check properties of transitions and runs.

Example

Consider the TSS MPA of Table 1. We can animate a transition for the term `a ; (( a ; delta) + delta )` as follows.

```
Maude> rew in TSS-Animation :
  ( MPA |- ( a ; ( a ; delta + delta ) == a ==> ) ) .
rewrite in TSS-Animation : MPA |- a ; (a ; delta + delta) == a ==> .
rewrites: 13 in 0ms cpu (0ms real) (~ rewrites/second)
result T: a ; delta + delta
```

7 Conclusions and Future Extensions

In this paper, we presented an initial attempt to implement SOS meta-theory in Maude. Our implementation defines a basic SOS framework with constant labels and provides a way to check the premises of congruence and operational conservativity meta-theorems. Furthermore, it allows for animating SOS specifications.

Maude was a very convenient choice for our implementation. Particularly, the correspondence between rewrites and transitions simplified the translation from SOS to Maude. The reflective semantics of Maude was crucial in our implementation. We expect easier and more efficient implementations as the meta-level facilities provided by Maude improve gradually.

In order to turn this prototype into a full-fledged tool for SOS, we foresee the following possible extensions:

(i) Implementing the more general SOS frameworks and their corresponding meta-theorems: There are more general SOS frameworks that allow for terms as labels, multi-sorted and binding signatures. Implementing such frameworks increases the applicability of our tool. Furthermore, the meta-theorems we implemented in this paper are among the most simple versions of the available meta-theorems for congruence and operational conservativity. By extending the SOS framework to more general settings, implementing the more general meta-theorems such as those of [20,11] would be beneficial.

(ii) Generating sound and complete equational theories: A class of meta-theorems that we did not address in this paper concerns generating equational theories from SOS specifications (see [2], for example). These meta-theorems also have an algorithmic nature and can be implemented in our framework.

(iii) Generating natural language documentation (and possibly research papers!) from the specified semantics.

(iv) Automatically generating the term matching and substitution definitions: To check the congruence and operational conservativity meta-theorems, we used routines that extract function symbol definitions from a theory. Using similar routines we may automate the substitution and matching procedures and make the Maude code for SOS specifications even more compact.

(v) Building a graphical user interface and importing SOS specifications from a general input format (e.g., XML).

Acknowledgments

Steven Eker and Paco Durán provided prompt and informative answers to our several questions about Maude. Ana Sokolova gave useful remarks on a preliminary version of this paper. Insightful comments of the anonymous reviewers of SOS workshop are acknowledged.

References

[1] The Maude system. Available from http://maude.cs.uiuc.edu/.

[2] L. Aceto, B. Bloom, and F. W. Vaandrager. Turning SOS rules into equations. *I&C*, 111(1):1–52, 1994.

[3] L. Aceto, W. J. Fokkink, and C. Verhoef. Structural operational semantics. In *Handbook of Process Algebra, Chapter 3*, pages 197–292. Elsevier, 2001.

[4] J. C. M. Baeten and C. A. Middelburg. *Process Algebra with Timing*. EATCS Monographs. Springer, 2002.

[5] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *JACM*, 42(1):232–268, Jan. 1995.

[6] R. Bol and J. F. Groote. The meaning of negative premises in transition system specifications. *JACM*, 43(5):863–914, 1996.

[7] C. d. O. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Departamento de Informática, Pontifícia Universidade Católica de Rio de Janeiro, Brasil, 2001.

[8] C. d. O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Mapping modular SOS to rewriting logic. In *Proceedings of LOPSTR'02*, volume 2664 of *LNCS*, pages 262–277. Springer, 2002.

[9] C. d. O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In *Proceedings of AMAST'00*, volume 1816 of *LNCS*, pages 407–421. Springer, 2000.

[10] P. Degano, F. Gadducci, and C. Priami. A causal semantics for CCS via rewriting logic. *TCS*, 275(1-2):259–282, 2002.

[11] W. J. Fokkink and C. Verhoef. A conservative look at operational semantics with variable binding. *I&C*, 146(1):24–54, 1998.

[12] J. F. Groote. Transition system specifications with negative premises. *TCS*, 118(2):263–299, 1993.

[13] J. F. Groote and F. W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *I&C*, 100(2):202–260, 1992.

[14] P. H. Hartel. LETOS - a lightweight execution tool for operational semantics. *Software, Practice and Experience*, 29(15):1379–1416, 1999.

[15] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer, 2002.

[16] J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *TCS*, 96(1):73–155, 1992.

[17] J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. In *Proceedings of AMAST'04*, volume 3116 of *LNCS*, pages 364–378. Springer, 2004.

[18] P. D. Mosses. Exploiting labels in structural operational semantics. *Fundamenta Informaticae*, 60(1-4):17–31, 2004.

[19] P. D. Mosses. Modular structural operational semantics. *JLAP*, 60-61:195–228, 2004.

[20] M.R. Mousavi, M.J. Gabbay, and M.A. Reniers. SOS for higher order processes. In *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR'05)*, volume 3653 of LNCS, pp. 308–322. Springer-Verlag, 2005.

[21] M. Mousavi, M. Sirjani, and F. Arbab. Specification and verification of component connectors. Technical Report CSR-04-15, Department of Computer Science, Eindhoven University of Technology, 2004.

[22] G. D. Plotkin. A structural approach to operational semantics. *JLAP*, 60-61:17–139, 2004.

[23] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceeding of the ACM Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 1–16. Springer, 1985.

[24] A. Verdejo. Building tools for LOTOS symbolic semantics in Maude. In *Peoceedings of FORTE'02*, volume 2529 of *LNCS*, pages 292–307. Springer, 2002.

[25] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In *Proceedings of FORTE/PSTV'00*, volume 183 of *IFIP Conference Proceedings*, pages 351–366. Kluwer, 2000.

[26] A. Verdejo and N. Martí-Oliet. Two Case Studies of Semantics Execution in Maude : CCS and LOTOS. *Formal Methods in System Design*, 27(1):113–172, 2005.

[27] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Technical Report 134-03, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Sept. 2003.